ARO 24960.27-MA

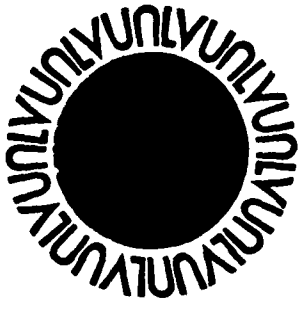
Grasp—A Graph Specification Language

Todd A Gross*
Dept. of Computer Science and Electrical Engineering
University of Nevada, Las Vegas

October 13, 1988

Report CSR-88-10

Department of Computer Science and Electrical Engineering



University of Nevada, Las Vegas
Las Vegas, Nevada 89154

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

MASTER COPY - FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

AD-A202524

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARO 24960.27-MA
6a. NAME OF PERFORMING ORGANIZATION Univ. of Nevada, Las Vegas 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code) Dept. of Computer Science & Electrical Engr. 4505 Maryland Parkway Las Vegas, NV 89154	7b. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING / SPONSORING ORGANIZATION U. S. Army Research Office 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAL03-87-G-0004
8c. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Grasp--A Graph Specification Language

12. PERSONAL AUTHOR(S) Todd Gross

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	October 1988	20

16. SUPPLEMENTARY NOTATION
The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Graph Theory, Specification Language, and Parallelization

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper is an introduction to Grasp, a language for defining and prototyping graph theoretic constructs and properties associated with them. The language is a specification language, which means that one gives only the necessary inputs and desired outputs, and the translator generates the necessary algorithm.

Section 2 explains why Grasp was devised. Section 3 gives the syntax of the language. Section 4 gives some examples of Grasp specifications. Finally, section 5 discusses the relative strengths and weaknesses of the language.

We are currently developing a translator from Grasp specifications into C functions.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

(2)

Grasp—A Graph Specification Language

Todd A Gross*
Dept. of Computer Science and Electrical Engineering
University of Nevada, Las Vegas

October 13, 1988

Report CSR-88-10

*Supported by the U. S. Army Research Office under Grant DAAL03-87-G-0004

Grasp—A Graph Specification Language

Todd A Gross
Dept. of Computer Science and Electrical Engineering
University of Nevada, Las Vegas

October 13, 1988

1 Introduction

This paper is an introduction to Grasp, a language for defining and prototyping graph theoretic constructs and properties associated with them. The language is a *specification language*, which means that one gives only the necessary inputs and desired outputs, and the translator generates the necessary algorithm.

Section 2 explains why Grasp was devised. Section 3 gives the syntax of the language. Section 4 gives some examples of Grasp specifications. Finally, section 5 discusses the relative strengths and weaknesses of the language.

We are currently developing a translator from Grasp specifications into C functions.

2 Purpose of Grasp

Originally, Grasp was intended to bring together two until now distinct fields of computer science: program synthesis and parallel computation. Due mainly to time constraints, we have pared the Grasp project to development of a translator from Grasp specifications to C functions. It is felt that later work could add parallel code generation and optimisation via program synthesis theory with only a minimum of rewriting of the present code.

The domain chosen for this research (and consequently for Grasp) is graph theory. There were several reasons for choosing this domain:

- It is an abstract domain, meaning there are relatively few details to keep track of. This facilitates synthesis.
- It is highly amenable to parallelization, as graphs are just sets of vertices and edges. Further, as graph theory asserts properties of graphs, rather than forces specific calculation[1], we can assume independence of

[1] Actually, in nonprocedural languages (like Grasp and PROLOG), calculation of results and assertion of properties are interchangeable paradigms. Thus a clause in PROLOG can be

calculation over the set of graph components. This greatly simplifies parallelisation, as we can forego dependency analysis [Wol88].

- It is a rich domain, including several problems that are simple to conceptualise but hard to calculate. One of these, the Travelling Salesman Problem, has already been used to test the power of specific parallel processing environments [KT88].

- It is a practical domain, as many real problems are at base graph theoretic. For instance, the topologies of multiprocessor networks are easily represented as graphs ([Hil85], Ch 3).

The present system, regardless of parallelism, is designed to allow one to define a nontrivial set of graph theoretic properties. It uses a small but powerful set of operators, as this is easier both to define and to use. Further, Grasp specifications are *nondeterministic*, which means they define a property but not how to compute it. This greatly facilitates parallel computation, because we are free to take advantage of all parallelisation inherent in the problem. Nevertheless, this must be left to later work.

3 Syntax

This section defines the set of legal specifications in Grasp, and illustrates the definitions with simple examples.

3.1 Identifiers

Identifiers are names that are bound to specific Grasp objects. Most objects in Grasp (or any other language, for that matter) have predefined names. For instance, 3 is a predefined name for the integer value 3. Only two types of objects in Grasp can (and must) be given names by the user: variables (§3.4) and definitions (§3.5).

Identifiers in Grasp must begin with a letter, and can otherwise consist of letters, digits, and underscores (_). They can be arbitrarily long, but if it's more than 32 characters long, any extra characters are right truncated. They also cannot be any of the 25 reserved words in Grasp. The language is case sensitive, and all reserved words are lower case. The following are legal Grasp identifiers:

```
i i0 i_1 inTEGER integers
```

The following are not legal:

```
0i  i integer
```

seen as calculating a set of valid answers or asserting which values would be logically consistent with the given axioms.

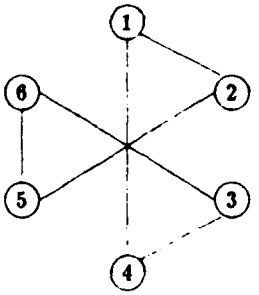


Figure 1: A simple graph

3.2 Types

There are 9 types in Grasp, of which 7 are graph-oriented (the other two being the standard types `integer` and `boolean`). In Section 1 we define the 9 types, including the syntax for literals in each type, and in section 2 we define classes of types that are used in defining operator syntax (§3.3), and give a hierarchy of the type classes.

3.2.1 Type Definitions

Standard Types There are two standard types in Grasp: `boolean` and `integer`.

The `boolean` type is exactly as in Pascal: there are two possible values, represented by the literals `true` and `false`.

Integer values are limited by the C compiler one runs the synthesized routines on, in our case $-2^{30}-1$ to 2^{30}. Note that while integers can take negative values, there are no literals to represent negative integers. This is because negative integers are rarely needed in graph theoretic problems. Thus `-3` is an illegal construct, unless preceded by something that evaluates to an integer. Integer literals are base 10 numerals, without any intervening symbols (including commas).

Graph Types To facilitate explaining the 7 graph types, we will use Figure 1.

Basic Components The basic components of graphs are vertices and edges. The corresponding Grasp types are `vertex` and `edge` respectively.

A `vertex` literal is represented by a period immediately followed by an integer. There are 6 vertices in the above graph, labelled 1 through 6. In Grasp these would be written as `.1`, `.2`,..., `.6`. Note that we are not forced to label vertices with consecutive integers, any 6 distinct nonnegative integers would do. For instance, if we wanted to use all primes we might label them `.2`, `.3`, `.5`, `.7`, `.11`, and `.13`.

An **edge** literal is represented by (v_1, v_2), where v_1 and v_2 are vertex literals. For instance, the long vertical edge between .1 and .4 would be represented by (.1,.4). Edges in Grasp are undirected (notice the lack of arrows in Figure 1), so (.4,.1) will also work, but Grasp converts this to the first form because it stores all edges in nondecreasing order.

In fact, (v_1, v_2) is a general form for edges, meaning that v_1 and v_2 can be any expression that evaluates to a vertex. For instance, if variable v is of type `vertex`, then (v,.2) is a valid edge form.

Sets of Components Grasp also lets one define sets of basic components—that is, vertex sets and edge sets. The corresponding Grasp types are `vset` and `eset` respectively.

Set literals are represented by { *list* }, where *list* is a nonempty set of either vertex or edge literals separated by commas. For instance, {.3, .5, .6} is a `vset` literal that represents a subset of the vertices in Figure 1. Again, the order is not important, so {.5, .6, .3} will work just as well. `Eset` literals work like `vset` literals, except with edges rather than vertices, so {(.3,.6), (.5,.6)} is a literal that contains all edges in Figure 1 with both vertices in {.3, .5, .6}.

Due to the impossibility of maintaining uniqueness of set elements (that is, preventing any elements of a set from repeating) at translate time, Grasp allows sets to contain any sequence of elements, as long as they are the right type. For instance, {.2, .1, .2} will be accepted by Grasp as a legal vset. We are planning to add a routine that will make sets proper, but this must wait for the rest of the translator to be constructed.

Note also that { *list* } is a general form for sets, so that {e1, (.1, v1)} is a valid `eset` form, given `e1` is an edge variable and `v1` is a vertex variable.

Sets of Sets When determining properties of sets of elements, it is often necessary to generate sets of sets. For instance, we say a set of vertices is *independent* if no two distinct vertices in that set have an edge between them. Thus, to determine if a `vset` is independent, we need to generate the set of all 2-element subsets of our vset. The Grasp types for these are `vsetset` for sets of `vsets`, and `esetset` for sets of `esets`.

Syntax of sets of sets is the same as for any other set, the only difference is the elements. A `vsetset` literal might look like this:

```
{ {.1} , {.1, .2} , {.1, .2, .3} }
```

There are 3 elements, each of which is a `vset` literal. The structure is analogous for `esetset` literals.

The general form for sets also works for sets of sets. Thus the following is a valid vsetset form, assuming `V` is a `vset`, and v1 and v2 are `vertex` variables:

```
{ V , {v1, .1} , {v2} }
```

Note that there is no type `vsetsetset` for sets of `vsetsets`. This is because such forms are only useful if one is interested in properties of sets of sets. The author

is not aware of any theories about sets of sets of vertices or edges, therefore there is no vsetsetset or esetsetset type. There is also not a general set extension because of difficulties with dynamic typing.

Graphs The final type in Grasp represents graphs themselves—which are, naturally, the main focus of graph theory. The corresponding Grasp type is **graph**.

A graph, mathematically speaking, is just a set of vertices and a set of edges on those vertices. In other words, a **vset** and an **eset**. The general form for a graph is { *vset* | *eset* } , where *vset* evaluates to a vset, and *eset* to an eset. The graph literal to represent the graph in Figure 1 is:

```
{ {.1, .2, .3, .4, .5, .6} | {(.1,.2), (.1,.4), (.2,.5),
  (.3,.4), (.3,.6), (.5,.6)} }
```

Note that, unlike with edges and sets, the order of placement in graphs *is* important. The vertexset must be to the left of the vertical bar, and the edgeset must be to the right.

3.2.2 Type Hierarchy

The 9 Grasp types are mutually disjoint, which means that no value can belong to more than one type. The form { } would appear to be a legal literal for any of the 4 set types, but it is not allowed in Grasp. Not only are the 9 types disjoint, they are also mutually incompatible, which means no value can be coerced from one type to another. For example, there is no automatic conversion of the vertex .1 to the vset {.1} .

Nevertheless, we can talk about a hierarchy of types, because certain operators will take values of more than one type. For instance, there is an operator to find the number of elements in a set. This operator will work on values of any of the 4 Grasp set types. We give 4 type classes that are used by Grasp operators, then draw a Hasse diagram of the type hierarchy.

The 4 type classes are:

smallset A set of graph components—a **vset** or an **eset**

setset A set of sets—a **vsetset** or an **esetset**

set Any set, thus either a **smallset** or a **setset**

element Any type that can be an element of a set, thus a **vertex** or an **edge**, or a **vset** or **eset**, as these are elements of their corresponding **setsets**.

We will use these names when we talk about types of operators and operands (§3.3). We draw the type hierarchy in Figure 2, where $x \rightarrow y$ means "expressions of type (class) x can evaluate to type (class) y.

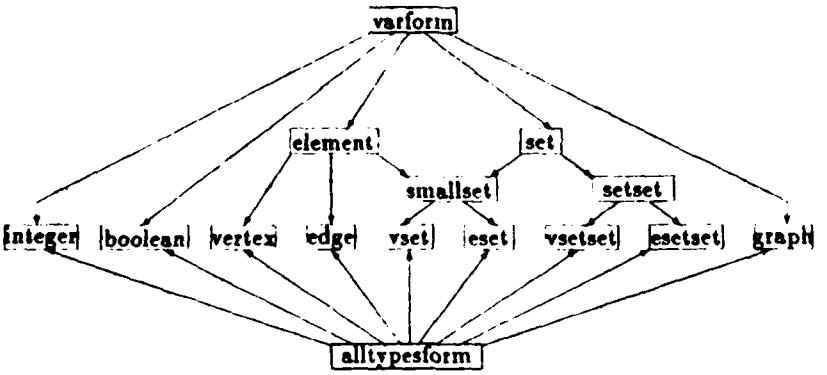


Figure 2: The type class hierarchy

You'll note there are two extra type classes in Figure 2: varform and alltypesform. A *varform* is any legal Grasp expression, regardless of which type it evaluates to. An *alltypesform* is an expression that can conceivably evaluate to any single Grasp type. In Grasp there are two such expressions: variables (§3.4) and definition use (§3.3.7). They were added to the diagram for the sake of completeness, it is assumed that the reader is familiar with this, at least on a syntactic level.

3.3 Operators

There are 21 operations one can perform in Grasp, each of which has its own operator(s). For each operation, we first give the syntax, where a type class name in italics means an expression that evaluates to that type class. The type class to the right of the arrow represents the return type class. Any expression may be enclosed in parentheses () and retain its type class.

3.3.1 Arithmetic

integer + *integer* ⟶ *integer*
integer - *integer* ⟶ *integer*

The only two arithmetic operations allowed in Grasp are addition (+) and subtraction (-). Arithmetic expressions are evaluated left to right, and both operators are strictly binary. For example, `5 - 3 - 1` evaluates to 1, and `- 3 - 1` is illegal. Parentheses can be used to override left to right evaluation, so `5 - (3 - 1)` evaluates to 3.

3.3.2 Relational

(*varform* = *varform*)	⟶	*boolean*
(*varform* /= *varform*)	⟶	*boolean*

In Grasp, one can only test for equality or inequality of two expressions, as "greater than" does not apply well to vertices or edges. While one can have any Grasp expression on either side of the relational operator, the expressions on both sides of the operator must evaluate to the same type. *vset* = *eset* won't work, even though both are elements of class smallset, and may even have the same syntax.

Notice that relational expressions are enclosed in parentheses. This avoids the problem of `a = b = c`, where `a`, `b`, and `c` are all boolean variables. One can enclose relational expressions in as many pairs of parentheses as one wishes, as long as one has at least one pair.

3.3.3 Logical

`not` *boolean*	⟶	*boolean*
boolean `and` *boolean*	⟶	*boolean*
boolean `or` *boolean*	⟶	*boolean*

Logical operators in Grasp are just like in Pascal. `and` and `or` evaluate left to right, with no guarantee of early evaluation (that is, `false and x and y` may or may not evaluate *x* and *y* though the expression will evaluate to `false` in any case), and `and` has higher precedence than `or`.

3.3.4 Graphical

`first` *edge*	⟶	*vertex*
`last` *edge*	⟶	*vertex*
`vset` *graph*	⟶	*vset*
`eset` *graph*	⟶	*eset*

Curiously, out of 21 operations available in Grasp, only 4 specifically operate on graphs or graph components—the rest operate on integers, booleans, sets, or definitions. And there is no operator whose sole operands are vertices.

`first` and `last` take an edge and return one of the component vertices. For instance, `first (.1, .3)` returns `.1`, and `last (.1, .3)` returns `.3`. Remember, though, that edges are put in nondecreasing order by the Grasp translator, so that `first (.3, .1)` will also return `.1`.

`vset` and `eset` take a graph and return the component vertex set and edge set respectively. Given a graph `G = { {.1, .2} : {(.1,.2)} }`, `vset G` evaluates to `{.1, .2}` and `eset G` to `{(.1,.2)}`. `vset` and `eset` are also the names Grasp uses for the corresponding types, but the translator can discern which use is intended (see (§3.2.1)).

3.3.5 Set Oriented

# *set*	⟶	*integer*
(*element* in *set*)	⟶	*boolean*
max *setset*	⟶	*smallset*
min *setset*	⟶	*smallset*
null *settype*	⟶	*set*
variable of *set* st *boolean*	⟶	*set*
subsets_of *smallset*	⟶	*setset*

In the above description, when *element* and *set* appear in the same statement, the types of the corresponding values must correlate. For instance, if the *set* expression evaluates to type `vset`, the *element* expression must evaluate to type `vertex`. The same rule applies between *smallset* and *setset*. For instance, if the *smallset* expression evaluates to type `eset`, the *setset* expression must evaluate to type `esetset`.

Size operator The first operator, #, is the *size* operator. That is, it tells you how many elements are in a set. For instance,

```
# {(.1,.4) , (.2,.3)}
```

evaluates to 2, as there are 2 edges in the eset, but

```
# { {(.1,.4), (.2,.3)} }
```

evaluates to 1, as there is 1 eset in the esetset.

Element operators The next three operators are *element* operators, which means they operate on individual elements of the set. The first of these is the in operator, which discerns whether a given element is in a given set[2]. Notice that the expression is in parentheses, it was necessary to enclose in expressions in parentheses to keep our grammar LALR(1)[3]. For instance, `((.2,.1) in {(.1,.2), (.2,.3)})` evaluates to `true`. But note that in only evaluates 1 level, so that `(.2 in { {.1} , {.2, .3} })` (*verter* in *usetset*) will not work.

The other 2 operators, max and min, take a set of sets, count the number of elements in each individual set, and return the set with the greatest and fewest number of elements respectively. If there is more than one largest (or smallest) set, it returns the first one it finds, which is not guaranteed to be the first one in the set. For example, if `S = { {.1} , {.1, .2} , {.1, .2, .3} }`, then `max S` evaluates to `{.1, .2, .3}` and `min S` evaluates to `{.1}`.

[2] Analogous the Pascal in operator.

[3] Later versions of Grasp should fix this kludge.

Null set operator The next operator, `null`, defines a null set of type *settype*, which can be any of the 4 Grasp set types. Thus, for example, `null vset` defines an empty vertex set.

Subset operator The next operator, the `of...st` operator, is a *subset* operator, which means it generates a subset of the set given to it (as the *set* operand just after the word `of`). It generates a subset by selecting those elements that satisfy a boolean expression (given after the word `st`)[4]. Let me give an example:

```
V of { {.1} , {.1,.2} } st # V = 1
```

Assume `V` is a variable of type `vset`. The set we're taking a subset of is a vsetset with 2 elements. Each element is bound to `V`, then tested to see if its size equals 1 (`# V = 1`). In this case, the test holds on the first element, but not on the second, so the expression evaluates to a *set* containing the first element, or { {.1} }.

Notice that the first operand is a *variable* of the proper element type. It exists only to bind individual elements of the set to be evaluated by the boolean function. Everywhere the operand appears in the boolean expression, it is replaced by an element of the set operand. But there is nothing that requires one to use this operand. In particular:

variable `of` *set* `st true`

evaluates to *set*, and

variable `of` *set* `st false`

evaluates to the empty set of that specific type. This, by the way, is a way of generating an empty set, although the `null` operator is the more preferable means of generating a null set. Mostly, this form is used in quantificational expressions (§3.3.6), but it can be used anywhere one wants to generate a subset.

Power set operator The last operator, the `subsets_of` operator, generates all subsets of a set—or if you prefer, the *power set* of a set. Clearly, the base set must be a smallset, as we will generate a set of sets.

Earlier, we gave a definition of an independent set of vertices; and said that we would have to generate all 2-element subsets of the vset to determine if it's independent. Letting `V` be our vset, and `V2` be a `vset` variable, we can generate the 2-element subsets with this Grasp expression:

```
V2 of subsets_of V st # V2 = 2
```

This expression is the same as the one above, only we've replaced a vsetset literal with a power set expression and changed the size of the desired vsets from 1 to 2.

[4] `st` stands for such that

3.3.6 Quantificational

(**forall** *subsetexp*) [*boolean*] ⟶ *boolean*
(**exists** *subsetexp*) [*boolean*] ⟶ *boolean*

Quantificational expressions are the most complicated ones in Grasp, but they are also the most useful: they allow the user to determine complex properties of graphs using tools they are likely to be already familiar with. The word *subsetexp* in the above syntax definitions refers to expressions using the **of...st** operator.

The expression takes the variable the **of...st** expression used to bind elements of the set operand and binds elements of the subset to it. This is then substituted into the boolean expression enclosed in brackets, one at a time. What the entire expression evaluates to depends, naturally, on whether the key word is **forall** or **exists**.

If the key word is **forall**, then the expression evaluates to **true** iff the boolean expression evaluates to **true** for each element of the subset, otherwise it evaluates to **false**. If the keyword is **exists**, then the expression evaluates to **false** iff the boolean expression evaluates to **false** for each element of the subset, otherwise it evaluates to **true**.

An example will help:

```
(forall v of {.1, .2, .3} st true)[v = .1]
```

First, look at the *subsetexp*. We want to generate the set of all elements in **{.1, .2, .3}** such that **true** is **true**. As you'll recall from (**§3.3.5.Subset**), this gives us back our original set. So our *subsetexp* is **{.1, .2, .3}**. Now we take each element of this set, bind it to **v** (assuming **v** is a **vertex** variable), and evaluate the boolean expression **v = .1**. For the first element, this evaluates to **true**, but for the other two it evaluates to **false**. Since it doesn't evaluate to **true** for every element in the subset, the entire expression evaluates to **false**.

But notice what happens when we replace **forall** with **exists**:

```
(exists v of {.1, .2, .3} st true)[v = .1]
```

Again, for the first vertex, the boolean expression evaluates to **true**, and for the second and third it evaluates to **false**. Since they didn't all evaluate to **false**, the entire expression evaluates to **true**.

Now for a more complicated example. Remember that we defined a vertex set to be *independent* iff for every pair of distinct vertices in the vertex set there are no edges between them. This is the Grasp equivalent of our definition:

```
(forall v1 of V st true) [
        (forall v2 of V st v1 /= v2) [
                not ((v1,v2) in eset G)
        ]
]
```

We will study this example more closely in Section 4, but for now notice that we have a *nested* quantificational expression. The syntax is important, the entire quantificational expression must go inside the brackets.

To evaluate this, first we generate the outer subset, in this case our input vset `V` and bind a specific element of it to `v1`. Using this binding, we evaluate the inner expression. We generate the inner subset, in this case `V` minus vertex `v1`, and bind an element of *that* set to `v2`. Then we evaluate the expression in the innermost brackets, with both bindings in effect. In effect, we will generate all pairs of vertices (v_1, v_2) such that v_1 and v_2 are in `V` and distinct from each other. Thus, using nesting, we can have as many bound variables as we like.

3.3.7 Definition Use

defnid (*arglist*) ⟶ *deftype*

Definitions are the basic components of Grasp specifications, just as functions are the basic components of C programs. And as one *calls* a function in C, one *uses* a definition in Grasp. We will give the syntax of definitions in (§3.5). For now we stick to using a definition: *defnid* is an identifier bound to a definition, *arglist* is a nonempty list of arguments to the definition separated by commas, and *deftype* is the type of the value the definition generates.

For instance, let's assume we've created a Grasp definition of an independent set. Our definition requires two pieces of information: the `vset` we're testing for independence, and the `eset` where any edges between our vertices may lie. Let's assume this information is stored in varibles `V` and `E` respectively, let's also assume the definition is named `independent`. Then we can use our definition on `V` and `E` by writing

```
independent(V,E)
```

3.4 Variables

Variables in Grasp are identifiers that are bound at any one time to an arbitrary value of a predefined type. In Grasp, three operations can be performed on a variable: *declaration*, *binding*, and *use*.

Declaration binds an identifier to a storage location and a concomitant type. A variable must be declared before it can be bound or used, and it cannot be redeclared in the same definition (§3.5). Thus the scope of a variable is from the point of declaration to the end of a definition. The syntax for declaration is

"*typename identifier*"

where *typename* is one of the 9 Grasp type names given in (§3.2). For instance, "`vertex v`" declares a vertex variable and associates the identifier `v` with it. Variables can be declared anywhere in a definition, as long as its name has not previously occurred in the definition. But there is no declaration statement, it is done at the first occurrence of the identifier in the definition.

Binding assigns a value to a variable. In fact, only subset and quantificational expressions bind variables—where elements of sets are bound to a variable for application to a boolean expression. In particular, it should be noted that there is no assignment operator or statement in Grasp. In this way, Grasp variables are not like variables in procedural languages like C or Pascal. Typically, Grasp variables are declared and bound at the same point in the specification.

Use binds the value associated with a variable to an expression. For example, suppose that the value associated with variable x is .3. Then, in the boolean expression `x in {.1, .2}`, we use x to get the expression `.3 in {.1, .2}`, which can be evaluated. Until now, every occurrence of a variable in a sample expression has been a use or binding. In the next section, we will start giving the proper declarations before using or binding a variable.

3.5 Definitions

Definitions are the basic components in Grasp. Just as a C program consists solely of a set of functions (plus global definitions), a Grasp specification consists solely of a set of definitions. Like functions, definitions have a set of input parameters and return a value. Like functions, definitions can be invoked by giving their name followed by a parenthetically enclosed set of arguments.

But unlike functions, definitions do not *do* anything. They only specify the desired output for a set of inputs, it is up to the Grasp translator to generate functions that can generate the desired outputs.

The syntax of a definition is

id **(** *arglist* **-->** *type* **)** **:** *property*

where *id* is the identifier bound to a definition, *arglist* is the set of input parameters, *type* is the type of the value the definition evaluates to (like a return value, except we're not "return"ing), and *property* is an expression evaluated using the variables in *arglist*.

Let's start with a simple example:

```
vcount("graph G" --> integer) :  #(vset G)
```

This is a definition of the number of vertices in a graph G. The name of the definition is vcount, there is one input to the definition G—which is the graph we're counting the number of vertices of. We say that vcount is a defintion *over* G. Notice that, since this is the first appearance of G in the definition, we have to declare it. In fact, this will be true of every input to a definition.

Definitions, like functions in procedural languages, generate one output. Since the number of vertices in a graph is an integer, the output to vcount is of type integer. We say vcount *evaluates* to an integer, the type our definition evaluates goes after the --> token and before the right parenthesis.

Then, after the colon, we have the actual definition: a Grasp expression that evaluates to the proper type (presumably, though not necessarily, using the inputs to the definition). To distinguish the expression that embodies the

definition from the entire definition, including the input and output, we sometimes call it a *property* of its inputs. Thus, in the case of vcount, #(vset G) is a property of G. The property can use any of the 21 Grasp operations, in any combination where operator/operand types match, but it must include at least one. Thus the following silly definition would be illegal:

```
[* three illegally defines 3 to be a property of graph G *]
three("graph G" --> integer):  3
```

But the following equally silly definition is legal:

```
[*
  new three legally defines 3 to be a property of graph G
*]
new_three("graph G" --> integer):  2 + 1 [* Uses the + operator *]
```

Notice that the preceding definitions had *comments*. Comments in Grasp are like comments in C, except that the delimiters are [* and *] instead of /* and */.

4 Examples

We now give some examples of Grasp specifications, which show the full range of expression in the language, and also its application to well-known and important graph-theoretic problems.

4.1 Independent Set

Our first example is a correct Grasp definition of an independent set:

```
[*
    independent() defines the concept of a set of vert-
    ices V being independent with respect to a graph G
*]

independent("vset V", "graph G" --> boolean) :
(forall "vertex v1" of V st true) [
        (forall "vertex v2" of V st v1 /= v2) [
                not ((v1,v2) in eset G)
        ]
]
```

This is the definition we gave in (§3.3.6), with the proper declarations and definition header. We used the same name as in (§3.3.7), but you'll notice that instead of an eset input we have a graph input. The reason is twofold: an eset need not be associated with a graph, but we want to know if a vset *in a*

particular graph is independent—that is, if any edges *in the graph we took the vertices from* straddle any two of our vertices.

Let's look at the definition line by line. Lines 1–4 are a comment documenting the definition. Line 6 is the definition *header*, and gives the information any other definition will need to know to use it. It gives the definition name, the two input paramaters, and the result type. Notice we have more than one input parameter. As in most programming languages, when we use this definition we must give the arguments in the same order as the parameters.

Lines 5–9 constitute the *body* of the definition. The body (or if you prefer, property) is a nested quantificational expression. Lines 5 and 6 bind variables `v1` and `v2` to different elements of `V`: line 5 binds `v1` to an arbitrary element of `V` and line 6 binds `v2` to any element that isn't the same as `v1`. Since each variable is eventually bound to every member of its corresponding subset, these 2 lines generate the set of all nonequal pairs of vertices. Line 7 takes each pair of vertices after binding and tests whether the corresponding edge exists in `G`. This definition will evaluate to `true` only when no such edge exists in `G` for *any* pair of distinct vertices in `V`. Which correlates to our English definition of an independent set of vertices: that there be no edges between any two of them.

4.2 Vertex Degree

According to [BM76], the *degree* of a vertex v is "the number of edges of G incident with v, each loop counting as two edges." Then in the Figure 3, `.1` has a degree of 3, and `.2` has a degree of 1.

Figure 3: Graph G

A Grasp definition of vertex degree looks like this:

```
[*
    The degree of a vertex v with respect to a graph G
    is defined as the number of edges in G incident to
    v, counting loops twice.
*]

degree("vertex v", "graph G" --> integer) :
    #("edge e1"  of  eset G  st  (v = first e1)) +
    #("edge e2"  of  eset G  st  (v = last e2))
```

First of all, notice that this definition generates an integer. Although most definitions will generate boolean values (as we are usually interested in whether a property holds for a given graph (component)), many generate other values. It is even possible to generate graphs, although the syntax is quite clumsy.

Lines 8 and 9 constitute the body of the definition. Notice that they are very similar to each other, the only differences are the names for the edge variables and the operations performed on them. We could have used the same variable on both lines, but we wanted to show that each line generates a distinct subset, even though both use `eset G` as their base. Line 8 generates the set of all edges that have `v` as its first component, then counts how many there are. Line 9 counts how many edges have `v` as their *last* component. These counts are then added together to get the incidence count, i.e. the degree. Although the following definition *looks* like it would work (and is perfectly legal), it doesn't count loops twice:

```
degree("vertex v", "graph G" --> integer) :
    #("edge e"  of  eset G  st  ((v = first e) or (v = last e)))
```

4.3 Path Between Vertices

An important concept in graph theory is the notion of a *path*, a set of edges such that given a *specific* list of vertices, there is an edge containing the n^{th} and $n + 1^{th}$ vertices in the list, so long as n is a positive integer less than the number of vertices in the list. For instance, given the eset {`(.1, .3), (.2, .3)`}, we can create the vertex list `.1 .3 .2`. There is an edge between `.1` and `.3` (`.1, .3`), and an edge between `.3` and `.2` (`.2, .3`). Figure 4 is a picture of the path.

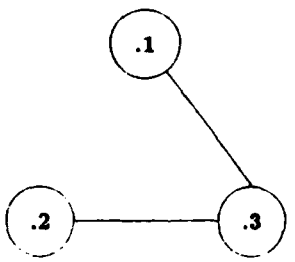


Figure 4: The path {(.1,.3), (.2,.3)}

And this is a Grasp definition of a path between two vertices:

```
[*
    A path to vertex v2 from vertex v1 exists in graph G iff one
    can create a list of vertices in G such that v1 is the first
    vertex, v2 is the last vertex, and any two consecutive vertices
    comprise an edge in G.
*]
```

```
path_to("vertex v1", "vertex v2", "graph G" --> boolean) :
        ( (v1,v2)  in  eset G )  or
        (exists "vertex v" of vset G st ( (v1,v) in eset G) )) [
                path_to(v, v2, {
                                "vertex w"  of  vset G  st
                                    (w /= v1)
                               |
                                "edge f"  of  eset G  st
                                    not((v1 = first f) or (v1 = last f))
                               }
                        )
        ]
```

Notice that the definition is *recursive*, which is to say it uses itself. Notice though, that each time we use the definition, the graph gets smaller: we remove a vertex from its vset, and all edges containing that vertex from its eset. Eventually we will either have (v1,v2) as a legal edge in G or we will run out of edges that are connected to v1.

The basic structure of this definition is: there is a path from v1 to v2 if either (a) (v1,v2) is an edge in the graph or (b) there is another vertex v in the graph such that (v1,v) is in the graph and there is a path from v to v2. The problem with the definition in this form is that it doesn't force us to make progress. If (v1,v3) were an edge in G, then path_to(v1,v2,G) could use path to(v3,v2,G) (substituting v3 for v) which could then use path_to(v1,v2,G) (substituting v1 for v), which takes us back where we started. This is why we reduce the graph with each use. Lines 11-17 generate the reduced graph by using the general form for graphs[5].

Earlier we mentioned that edges in Grasp are automatically canonized. The main reason this is done is to make writing specifications easier. Note, for instance, that we wrote (v1,v2) in eset G but not (v2,v1) in eset G in the definition of path to. We didn't have to, even though we don't know (and can't know in general) whether v1 or v2 will appear first in the edge. This is the advantage of canonizing edges: we never have to write a separate expression for each possible edge form.

4.4 Connected Graphs

We say that a graph is *connected* if there is a path from every vertex in the graph to every other vertex in the graph. Or more simply, if there is only one "piece" in the graph. All 3 graphs we've drawn so far have been connected graphs, figure 6 shows a graph that isn't connected, as there's no path from .1 to any other vertex.

Now we give the Grasp definition:

[5] This, by the way, is the clumsy syntax I alluded to earlier.

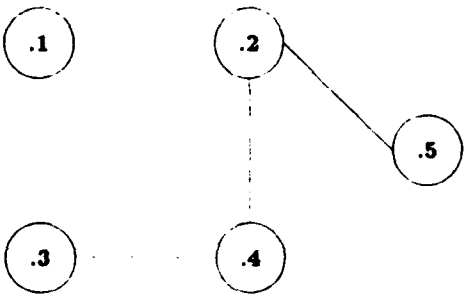


Figure 5: A nonconnected graph

```
[*
    A graph is defined to be connected if there is a path from
    every vertex to every other vertex in the graph.
*]

connected("graph G" --> boolean) :
        (forall  "vertex v1"  of  vset G  st  true) [
               (forall  "vertex v2"  of  vset G  st  (v2 /= v1)) [
                      path.to(v1,v2,G)
               ]
        ]
```

Notice the similarity between this definition and our definition of `independent`. Both operate over pairs of vertices, but `connected` tests for paths between vertices, and `independent` tests for a lack of edges between vertices (which, by the way, doesn't mean that there can't be paths between the vertices). Notice also that we used the definition of `path to` in this definition—we used it to define a path between two vertices.

By now, one should be familiar enough with Grasp syntax to be able to read relatively simple definitions (like `connected` above) and be able to ascertain their meaning.

5 Conclusions

We have not yet produced a Grasp translator, so the conclusions we reach are based on the inherent capabilities (or the lack thereof) of the language. In trying to define properties of graphs using Grasp, we have found it to be a surprisingly powerful language. If one can come up with a systematic way of defining a property, it appears one can use Grasp to define it. This is not to say that it is always *easy* to define such a property, we have found that concepts that naturally belong together sometimes need to be separated in the definition. For

instance, if we want to say that an edge contains a vertex, we have to say it's either the first component or the last one.

There are two main problems with Grasp as a language. The first is a lack of special-purpose operators. For example, when defining path_to, it would have been easier if we had an operator that subtracted a vertex from a graph as defined in ([BM76] (§1.4)). But as long as the language has a good general set of operators, it was felt that these operators could be added later.

The second problem is more serious. The calculation involved in generating a graph property appears to be enormous, because we tend to have to generate and test all possible operands. For instance, in path_to, given a graph G, we will generate $G - v$ for all vertices v in G. And for each of *these* graphs we generate $G - v - v'$ for all vertices v' in $G - v$, and so on. That's $O(n!)$ graphs we have to generate! It would be much cheaper to use transitive closure to determine which vertices are connected, but Grasp doesn't have matrix operations. For now, we are content to have a general-purpose experimental language.

But even with its apparent faults the language holds great promise. In [Bal87], Douglas Baldwin discusses the failure of current programming languages to effectively handle parallelizable problems. He cites four problems in particular, each of which would be handled by the complete Grasp system:

1. **Data Dependencies** Because the user cannot assign values to variables or determine order of execution, he or she cannot create data dependencies. Thus we are free to find and utilize all available parallelisation.

2. **Data Parallelism** As we've said before, graphs are *merely sets of vertices* and edges, all of which are very similar to each other. Therefore, a high degree of data parallelism will be inherent in a typical Grasp specification.

3. **Granularity** Because there are many parallel operations over sets of highly similar data in the typical Grasp specification, and one has complete data independence, one is free to divide the problem into components of very specific number and size. Granularity can closely match the specific hardware and/or software one is operating under.

4. **Generality** Grasp is not a general purpose language. But it generates C functions, thus it is possible to use the general purpose capabilities of C. In particular, it is possible to call external C functions using definition use syntax[6].

It is hoped that further work will extend the present system to include parallelism as originally intended.

[6] Grasp defintions will be translated directly to C functions, and definition uses to C function calls. So as far as the C compiler is concerned, they are completely interchangeable.

References

[Bal87] D Baldwin. Why we can't program multiprocessors the way we're trying to do it now. Technical Report 224, University of Rochester, Rochester, NY, Aug 1987.

[BM76] J A Bondy and U S R Murty. *Graph Theory with Applications*. North-Holland, New York, 1976.

[Hil85] W D Hillis. *The Connection Machine*. PhD thesis, MIT, Cambridge, MA, 1985.

[KT88] M Kallstrom and S S Thakkar. Programming three parallel computers. *IEEE Software*, 5(1):11–22, 1988.

[Wol88] M Wolfe. Multiprocessor synchronization for concurrent loops. *IEEE Software*, 5(1):34–42, 1988.